A LANGUAGE INDEPENDENT DEBUGGER SEMANTICS BASED DEBUGGING IN $\mathbb{K}$

BY

MANASVI SAXENA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Prof. Grigore Roşu

ABSTRACT

This works presents the $\mathbb{K}$ debugger - a language independent program debugger. The debugger is a part of the suite of tools that form the $\mathbb{K}$ framework. Conventional language dependent debuggers rely on an ad-hoc model of the underlying programming semantics, and may thus be incapable, or inaccurate in their ability to rectify a program's behavior. The $\mathbb{K}$ debugger uses a different approach - it's parametric over the $\mathbb{K}$ semantics of the programming language, which exposes accurate and subtle faults. The $\mathbb{K}$ debugger generalizes behaviors of conventional debuggers, providing users with a uniform interface that works across programming languages. Moreover, the $\mathbb{K}$ debugger is formal, performant and highly configurable, allowing it to adapt to the any programming language. This makes the $\mathbb{K}$ debugger a suitable replacement to traditional language specific debuggers.

TABLE OF CONTENTS

Chapter 1: INTRODUCTION

Traditional program debuggers, such as GNU C's gcc [1] and Java's JDB [2] are established tools for debugging programs. However, these language specific tools are generally based on ad-hoc semantics of the target programming language, and have unclear semantics themselves. For instance, consider the case of gcc, one of the most widely used tools for debugging C. Since it's not directly derived from the official C standard [3], the semantics of the tool itself are unclear. For instance, it's not well defined what the definition of the gcc's "step" command is. The deficiencies in the clarity of the semantics can sometimes lead to very subtle bugs creeping into programs. In gcc's case, there are multiple other tools such as valgrind [4], RV-Match [5, 6] e.t.c. that help in dealing with these subtle bugs, such as undefined behavior.

This work presents the $\mathbb{K}$ debugger: a formal semantics based language independent alternative to traditional debuggers [1]. Our approach works for all programming languages, as long as the semantics of the target language have been captured formally in the $\mathbb{K}$ framework. Apart from being language independent, our approach is formal. For instance, the $\mathbb{K}$ debugger, which forms the main contribution of this work, has well defined semantics for all of its commands. Since our approach is parametric over a language's operational semantics, it can expose very subtle bugs, that traditional debugger miss. Moreover, our approach is performant and practical, and has been used in practical non-trivial settings (see chapter 7).

1.1 OUTLINE

The basic outline of this works is as follows:

- Chapter 2 presents preliminaries, mainly focused on the $\mathbb{K}$framework, its philosophy and the underlying logical foundations.
- Chapter 3 presents an overview of the $\mathbb{K}$ debugger.
- Chapter 4 introduces details about the information stored in the debugger as part of its state.
- Chapter 5 introduces the challenges observed while implementing the debugger, and the solutions.

[1] The $\mathbb{K}$ debugger's code is available as a part of the entire framework at `https://github.com/kframework/k`

- Chapter 6 details the debugger's architecure in context of $\mathbb{K}$, and the its code.
- Chapter 7 presents a concrete use case of the debugger. It outlines how the debugger can be used to develop and debug both language semantics and programs in a target language.
- Chapter 8 discussed the future work, and provides closing remarks

Chapter 2: PRELIMINARIES

2.1 THE $\mathbb{K}$ FRAMEWORK

$\mathbb{K}$ is a logical framework for defininining programming languages and type systems [7]. The $\mathbb{K}$ framework has been successfully used to capture semantics of many programming languages such as C [8], Java [9], Javascript [10] and the EVM [11] (Ethereum Virtual Machine)

This section presents a brief overview of $\mathbb{K}$, which forms the basis of contributions presented in this work.

2.1.1 Overview

Broadly, $\mathbb{K}$ is a rewrite based executable semantics framework. Semantics to programming language constructs, in $\mathbb{K}$, are defined using "configurations" and "rules". A $\mathbb{K}$ configuration, in simple terms, is a bag (multiset) of nested cells. A $\mathbb{K}$ cell is a unit that contains information relevant to program execution [12].

Semantics in $\mathbb{K}$ are defined using computation rules that operate over configurations. $\mathbb{K}$ rules are written using a simple notation that resembles small-step execution semantics of the target language. The syntax of $\mathbb{K}$ is made to allow writing small step semantics easier. Once the semantics of the target language have been captured, the framework uses the semantics to provide a suite of tools, which include an interpreter, a symbolic execution engine, a program verifier , and a debugger. The $\mathbb{K}$ debugger is the main contribution of this work, and is discussed in detail in the reamainder of this work. This semantics first approach is central to the framework [13]. Instead of relying on language specific tools, $\mathbb{K}$'s formal semantics based philosophy not only automatically generate tools, but also provides a formal logical framework (discussed in 2.2) as their foundation. The remaining sections in this chapter focus on the providing greater depth into aspects of the framework that serve as a precusor to understanding the rest of this work.

2.1.2 $\mathbb{K}$ Syntax

As mentioned earlier, $\mathbb{K}$ has been successfully employed to model real world programming languages. In order to emphasize the abstractions and abilities of the framework and the debugger, we chose to use an actual real-world language as an example, instead of a toy

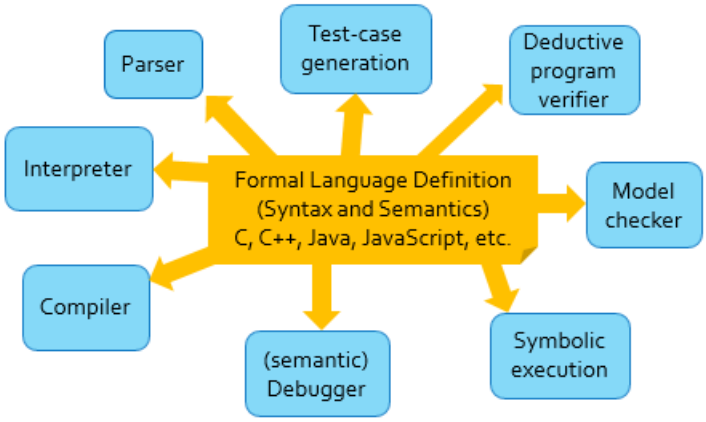


Figure 2.1: $\mathbb{K}$'s Semantics First Approach

language. The language used to drive the content in this, and the following chapters is the EVM (Ethereum Virtual Machine). The EVM is a stack-based assembly like langauge that's used for programmble transactions on the Ethereum Blockchain. Here, we describe what $\mathbb{K}$'s syntax is like using EVM as an example [14]

2.1.3 Language Configurations

The "configuration" keyword describes the layout of the environment that a program in the language would execute in. For program execution, a particular instance of the configuration represents the state of the program. In $\mathbb{K}$, the configuration is described in an "XML-like" notation, and contains nested cells. Consider for example, a segment of the configuration of the EVM semantics -

Example 2.1.

```
configuration <k> $PGM:EthereumSimulation </k>
              <evm>
               <id>        0          </id>
               <program>   .Map       </program>
               <pc>        0          </pc>
               <wordStack> .WordStack </wordStack>
               <localMem>  .Map       </localMem>
               <gas>       0          </gas>
               ...
              </evm>
              ...
```

In example 2.1, the keyword "configuration" marks the beginning of the configuration declartion. The cells have values that they'd be initialized with at the start of a program's execution. For instance, the `<pc>` cell, which represents the program counter, starts with a value 0. The program on the other hand, is an empty map. During execution, the `<program>` cell holds key-value pairs, where the keys are program counter values, and the values are program instructions. The `<k>` cell has special meaning, it holds the program itself. In the example configuration above, the cell contains the `$PGM` variable, which during execution, will be replaced by the contents of the parsed program.

2.1.4 Language Syntax

The syntax of the programming language being defined must be captured in a "BNF" like notation. $\mathbb{K}$'s syntax is highly specialized to allow definition of language syntax easier. For instance, let's consider the syntax of simple EVM instructions.

Example 2.2.

```
syntax UnStackOp  ::= "SLOAD"
syntax BinStackOp ::= "SSTORE"
```

In example 2.2, the `SLOAD` evm instruction is parsed as a "Unary Stack Operation" (referred to in the semantics as "UnStackOp", while `SSTORE` is parsed as a "Binary Stack Operation". Stack operations in EVM represent simple instructions that push/pop values of the stack. Note that `SLOAD` and `SSTORE` are in the syntax of EVM itself. In section 2.1.5, we describe how parsed constructs are used in giving semantics to the language itself.

2.1.5 Language Semantics

Rewrite rules form the basis of defining language semantics in $\mathbb{K}$. Rewrite rules in $\mathbb{K}$ take the form $\alpha \Rightarrow \beta$, . α and β are patterns over $\mathbb{K}$ configurations. The rules capture the operational semantics of the language - they determine how a program in the language executes w.r.t. the semantics of the language. In order to better illustrate $\mathbb{K}$ rules, we use some simple rules from the $\mathbb{K}$ semantics of EVM.

Consider the following rule from the EVM semantics -

Example 2.3.

```
rule <k> SLOAD I => V ~> #push ... </k>
     <id> ACCT </id>
     <account>
       <acctID> ACCT </acctID>
       <storage> ... I |-> V ... </storage>
       ...
     </account>
```

The rule from example 2.3 operates over the `SLOAD` Unary Stack Operation Instruction describe in 2.1.4, and describes its operational semantics. This rewrite rule is a fairly complicated one, but we chose it as an example as it concisely captures the features of the framework that we'd like to highlight in this section. The rule, aptly written using the keyword `rule` describes the following actions:

- Recall that the `<k>` cell is populated with the parsed EVM program. This particular rule specifies that if the SLOAD instruction is present at the top of the cell, followed by an address, then
  1. Match the variable `I`, on the concrete address observed during program execution.
  2. Use the concrete value of the variable `I` as the key to lookup the mapped value (denoted by address `V`) at the given address.
  3. Put the value `V` at the top of the `<k>` cell, so that it can be pushed onto the stack via the `#push` instruction.
- `#push` is an internal instruction, i.e. it's not a part of the semantics, but is used a helper function for writing the semantics.
- The rule also accentuates features of $\mathbb{K}$ that make it an easy framework to define semantics in. Note the use of `...` construct in the rule. The `...` allow for mentioning only relevant part of the semantics in rule definition. The non-relevant parts of the configuration are inferred from context, which allows the rules to be compact, and easy to write and read. In order to understand the `...` better, notice in particular in `<storage>` cell. It's not possible to mention the entire storage for every program when writing the semantics, however, the `...` construct allows for mentioning only the part of the storage map that's relevant to the rule, making writing the rule concise.

- Furthermore, notice the use of the sequencing operator (denoted via the symbol `~>`. This is a special $\mathbb{K}$ operator that "sequences" computational tasks. In other words, it schedules another computation to follow the current computation. In the example above, the `#push` instruction is scheduled to happen after the `SLOAD` lookup computation performed by the current rule.

2.2 THEORETICAL FOUNDATIONS OF $\mathbb{K}$

This section briefly discusses $\mathbb{K}$'s theoretical foundations. At it's core, $\mathbb{K}$ performs Reachability Logic reasoning [15]. Reachability logic is a sound and relatively complete logic tailor-made for reasoning about reachability properties. Unlike program analysis techiniques, $\mathbb{K}$'s Reachability Logic based approach allows it to use "one canonical model" of the language, derived from it's operational semantics, as the basis for reasoning and analysis. This is a departure from traditional program reasoning techiniques like Hoare Logic [16], which rely on denotational semantics of a language for reasoning [17]. While denotational semantics may be easy to reason about, they're not suitable for use as basis of interpreters and debuggers (due to the granularity of the semnatics). $\mathbb{K}$ is able to solve this gap in semantic approaches using a convenient logic, and language independent tools based around the logic.

2.2.1 Reachability Logic

The general form of a $\mathbb{K}$ rule, written as $\alpha \Rightarrow \beta$. The rule notation in $\mathbb{K}$ is heavily rooted in Reachability Logic. The $\Rightarrow$ symbol, also read (reaches) is a logical connective in Reachability Logic. Semantic rules defined in the language become reachability axioms while reasoning in RL. In order to present a clearer picture of RL, the sound and complete proof system of the logic is given in figure 2.2. Note that $\mathcal{A}$ is the initial (trusted) execution semantics of the programming language (axioms). The $\mathcal{C}$ on $\vdash_{\mathcal{C}}$ indicates that the *circularities* $\mathcal{C}$ are reachability claims conjectured but not yet proved. The **Circularity** proof rule allows us to conjecture any to-be-proven reachability claim as a circularity, while **Transitivity** allows us to use the circularities as axioms (only after we have made progress on proving them).

$\mathbb{K}$ uses RL as the underlying logic to reason about reachability, but RL itself is parametric over a static logic required for reasoning about the "matching". $\mathbb{K}$ uses a section of Matching Logic (ML) as this static logic.

Case Analysis :

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi \qquad \mathcal{A} \vdash_{\mathcal{C}} \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

Axiom :

$$\frac{\varphi \Rightarrow \varphi' \in \mathcal{A}}{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi'}$$

Abstraction :

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi' \qquad X \cap \mathit{FreeVars}(\varphi') = \emptyset}{\mathcal{A} \vdash_{\mathcal{C}} \exists X\ \varphi \Rightarrow \varphi'}$$

Circularity :

$$\frac{\mathcal{A} \vdash_{\mathcal{C} \cup \{\varphi \Rightarrow \varphi'\}} \varphi \Rightarrow \varphi'}{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi'}$$

Reflexivity :

$$\mathcal{A} \vdash \varphi \Rightarrow \varphi$$

Transitivity :

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow^{+} \varphi_2 \qquad \mathcal{A} \cup \mathcal{C} \vdash \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi_3}$$

Logic Framing :

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi' \qquad \psi \text{ is a (patternless) FOL formula}}{\mathcal{A} \vdash_{\mathcal{C}} \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

Consequence :

$$\frac{\models \varphi_1 \rightarrow \varphi_1' \qquad \mathcal{A} \vdash_{\mathcal{C}} \varphi_1' \Rightarrow \varphi_2' \qquad \models \varphi_2' \rightarrow \varphi_2}{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi_2}$$

Figure 2.2: Sound and relatively complete proof system of Reachability Logic.

2.2.2 Matching Logic

Matching Logic [18] is the static logic used in $\mathbb{K}$ for reasoning about programs. Consider the general form of the $\mathbb{K}$ rule we've used throughout this work, written as $\gamma \equiv \alpha \Rightarrow \beta$. In this case, γ is a reachability claim in RL, usually used as an axiom in the RL proof system, while α and β are ML formulas (known as patterns). Broadly speaking, $\mathbb{K}$ the matching process in Reachability Logic can be formally justified using Matching Logic.

2.3 RUNNING $\mathbb{K}$

So far, this work has provided an introduction to the philosophy and theoretical foundations of $\mathbb{K}$. This section however, takes a slight diversion into the practical aspects of the framework, and describes briefly the process of running the framework itself.

$\mathbb{K}$ is an open source project under the UIUC/NCSA license, and the source is available at `github.com/kframework/k`. The online repository has installation instruction for entire framework. Once installed, the basic outline, assuming one is familiar with $\mathbb{K}$'s syntax is

as follows:

- The extension ".k" represents files containing $\mathbb{K}$ code.
- Usually, the syntax of the target language is place in a dedicated module, and the semantics are orgranized into a main module (which imports the syntax module).
- The framework is initialized using the `kompile` command. The command takes as input the main semantics module, and generates all tools provided by the framework. Errors with $\mathbb{K}$'s syntax are reported at this point.
- The `krun` command runs automatically generated tools, after the framework has been instantiated using the `kompile` command. In its default setting, the command takes as input a program in the target langauge, and runs interprets it using the semantics.
- The `krun` command also takes as input various flags, which are used to invoke other tools. This particular work focuses on the debugger, which is invoked using the flag `--debugger`. Once the flag has been passed to the tool, the framework drops the user into the "debug" mode, which will be described in detail in chapter 3.

Chapter 3: $\mathbb{K}$ DEBUGGER: AN OVERVIEW

3.1 INTRODUCTION

The $\mathbb{K}$ debugger, the main contribution of this work, is a language independent program debugger with strong theoretical foundations. It's provided as a part of the suite of tools generated via the $\mathbb{K}$ semantics of a language, when the language's model is captured in $\mathbb{K}$. This chapter goes into depth about the syntax and semantics of the debugger's commands.

3.2 COMMANDS SYNTAX

As mentioned in 2.3, passing the `--debugger` flag to $\mathbb{K}$ drops the user in the debugger mode. The debugger mode is intended to serve as a Read Evaluate Print Loop (REPL), with the program loaded. The debugger's REPL provides an experience similar to other debuggers, such as GNU C's gcc. Furthermore, it supports "tab" autocomplete, making it suitable for use as an alternative to conventional language specific debuggers.

The debugger's REPL is controlled using commands that have syntax tailor-made for ease of use. We present the syntax in JavaCC's BNF-like notation. For users unfamiliar with JavaCC [19], we provide a relation between JavaCC's syntax and BNF notation -

- Consider the JavaCC code for the step command, which looks like -

```
{ <STEP: (["s", "S"])| "step" >}
```

- This translates into BNF syntax -

```
syntax STEP ::= "s" | "S" | "step"
```

Figure 3.1 presents the JavaCC's BNF-like syntax for generating the debugger's parser.

3.3 COMMANDS SEMANTICS

3.3.1 Overview

This section presents the debugger's commands, and their functionality.

In table 3.3.1, a summary of the commands, and their usage is presented. The **Command Strings** column list the strings that the REPL accepts for the command. Some debugger

```
TOKEN: /*RESERVED TOKENS */
{
        <STEP: (["s", "S"])| "step" >
    |   <BACKSTEP:  (["b", "B"])| "back-step">
    |   <JUMPTO: (["j", "J"])| "jump-to">
    |   <QUIT: "quit" | "abort" | "exit">
    |   <CHECKPOINT: "checkpoint" | "ch">
    |   <RESUME: "resume" | "run" | "r">
    |   <PEEK: "p" | "peek">
    |   <REMWATCH: "remwatch" | "xwatch">
    |   <SHOW: "show"> : STRING_STATE
    |   <GETSTATE: "get-states" | "gs">
    |   <SELECT: "select">
    |   <SOURCE:"source" | "src"> : STRING_STATE
    |   <COPY: "copy" | "cp">
    |   <WATCH: "watch" | "w"> : STRING_STATE
    |   <NUMBER: (["0" - "9"])+>
    |   <STRING : (["a" - "z", "A"-"Z", "0"-"9"])+ >
    |   <NEWLINE : "\n" | "\r\n">
    |   <UNACCEPTED : (~[" "])+>

}

<STRING_STATE> TOKEN:
{
        <PATTERN : (~[])+> : DEFAULT
}
```

Figure 3.1: Syntax of the $\mathbb{K}$ Debugger

commands don't need any parameters (denoted via an empty **Parameters** cell in the table), while others either need optional parameters (denoted using []), or required/mandatory parameters (denoted using $\langle\rangle$).

3.3.2 Command Semantics

This section attempts provides theoretical bases to the debugger's commands.
We divide the commands into three broad categories -

- **Trace Exploration** - Commands that allow the user to explore the trace to localize the bug. Consider a program execution trace, denoted by $\tau \equiv \varphi_0 \Rightarrow \varphi_1 \Rightarrow \cdots \Rightarrow \varphi_i$. The state exploration commands are `step`, `resume`, `jump`, `backstep`. Assume n, the input

Command	Command Strings	Parameters	Functionality
Step	s, S, step	[No of steps]	Take the specified number of steps from current pattern
Backstep	b, B, back-step	[No of steps]	Take the specified number of steps backwards
Jump To	j, J, jump-to	⟨State Id⟩	Jump to a particular state in the current trace
Quit	quit, abort, exit		Exit debugger session
Resume	r, resume, run		Continue execution until final state is reached
Peek	p, peek		Print the current pattern
Checkpoint	ch, checkpoint	⟨Checkpoint Interval⟩	Set interval at which checkpoints are recorded
Show	show	⟨pattern⟩	Match the given pattern on the current state and print the result of simplifying the ML implication
Get State	gs, get-state	⟨State Id⟩	Print the state specified by the state id, without explicitly jumping to it
Watch	watch, w	⟨Watch Pattern p⟩	Add p as a "watch". Watch patterns are special patterns that are matched, and the result printed whenever the state is changed
Remove Watch	remwatch, xwatch	⟨Watch Pattern Id⟩	Remove the specified watches patterns
Source	source	⟨File Path⟩	Given a debugger file, interpret the file in debug mode

Table 3.1: Summary of $\mathbb{K}$ Debugger's Commands

from the user denoting the number of ML patterns to use in the exploration process, φ_i, the latest pattern observerd in the trace, and φ_f, the final state in the trace . The `step` command moves the state to φ_{i+n}, assuming that $i + n < f$. The `backstep` command moves the debugger to pattern φ_{i-n}, if $i - n >= 0$, or φ_0, if $i - n < 0$. The `jump` command moves the debugger to pattern φ_n, as long as $0 \leq n \leq f$, while resume moves the debugger to state φ_f.

- **State Modification** - Commands that modify the debugger's state. Note that the Trace Exploration commands also modify the state, however, unlike the State Modification commands, their primary purpose is to help the user explore the trace. These commands have compliex algorithms associated with them, as they keep track of the current trace. They're descirbed in detail in section 5.2

- **IO** These are commands that involve IO interaction. We omit describing them here, as a sufficient description is provided in 3.3.1

Chapter 4: $\mathbb{K}$ DEBUGGER STATE

4.1 INTERACTION WITH $\mathbb{K}$'S BACKEND

This chapter provides a brief introduction of the $\mathbb{K}$ debugger's "state management" strategy. As described in section 2.3, the `krun` command interprets the program passed to it as input, in a single shot. The $\mathbb{K}$ backend architecture allows for specifying an upper bound on the number of rewrite steps a term should execute for. In order to be efficient, the debugger maintains its own state to minimize rewrite calls to the $\mathbb{K}$ backend.

4.2 COMPONENTS OF DEBUGGER'S STATE

At any given time during the debugging session, the following state related information is stored. In the remainder of this section, `debuggerState` represents the state of the debugger, and `debuggerState.<component>` represents a particular component of the state. For instance,
`debuggerState.initialPattern` represents the "initial Pattern" observed in the trace, and recorded as part of the debugger's state.

- **Initial Pattern** - Let φ_0 represent the pattern encountered by the tool at the initial call to the debugger. The pattern is stored as a special "start" pattern in the debugger. A jump call to the debugger, with `stateId` to jump to specified as 0 results in the debugger entering a state where the current pattern is φ_0.

- **Visited checkpoint patterns** - The debugger keeps a track of certain patterns that are encountered during execution. These patterns are determined by the "checkpoint" interval, or the interval at which the patterns are recorded. Suppose, during the execution of a program, the following trace τ is observed, $\varphi_0 \Rightarrow \varphi_1 \cdots \Rightarrow \varphi_n$, where n is either the upper bound on the number of steps, or the total steps before the program terminates, then the debugger records the trace $\varphi_0 \Rightarrow \varphi_c \Rightarrow \varphi_{2*c} \Rightarrow \varphi_{\lfloor \frac{n}{c} \rfloor}$, where c is the checkpoint interval.

- **Watches** - All user defined watch patterns, which will be referred to as $\varphi_{w0}, \varphi_{w1} \ldots \varphi_{w_n}$ for the remainder of this work, are also stored as debugger state.

- **Current Pattern** - The debugger keeps a track of pattern φ_i, where φ_i is a pattern in the trace attained after a trace exploration command (see section 3.3.2). For instance,

if φ_a is the current pattern, where $0 < a < n$ in τ, then `backstep 1` will set the Current Pattern component of the state to φ_{a-1}.

Chapter 5: CHALLENGES AND SOLUTIONS

5.1 OVERVIEW

The $\mathbb{K}$ debugger is more general than traditional debuggers. The generality has many advantages, mainly -

- **Language Independence** - Unlike traditional debuggers, which have to be implemented for every target language, the $\mathbb{K}$ debugger is simply derived from the semantics. Our approach has direct advantages over traditional debugging techniques.

- **Initial Implementation Effort** - The only effort needed to make the $\mathbb{K}$ debugger work for a language is to define the semantics of the language in $\mathbb{K}$. However, in order for any programming language to be practically useful, it usually has to have at least parser, and interpreter. Usually, these tools are developed using informal language specifications, as in the case of C, EVM, Java, and others. Since defining the semantics in $\mathbb{K}$ provides these tools for free, it's a reasonable approach to start defining the language directly in $\mathbb{K}$, as in the case of IELE [20]. If this "semantics first" approach is followed, then there is no extra implementation effort for obtaining a working debugger for the target language.

- **Maintainance Effort** - As newer versions of a language are released, traditional debuggers have to be updated to reflect the changes in the language. Moreover, newer versions of the language may require a shift in the way programs are debugged. This may render existing ad-hoc debuggers useless. In the case of the $\mathbb{K}$ debugger, a change in the language semantics is reflected via a change in the executable specification. Since the debugger is parametric over the semantics, no change is required to maintain the debugger; it's automatically updated as the language's specs change.

- **Unified Debugging Experience** - The set of commands for the $\mathbb{K}$ debugger remains fixed **across languages**. Once a user becomes familiar with the debugger's syntax and semantics, it takes no extra effort to debug programs in different languages.

- **Formal Foundations** - The $\mathbb{K}$ debugger is parametric over formal SOS of the target language. Since the debugger is based on $\mathbb{K}$, all of the commands This allows fine grained semantics based debugging of programs in the language, leading to subtle bugs being caught.

This, we believe, renders the need for defining language specific debuggers pointless, as long as the $\mathbb{K}$ debugger can generalize the role of traditional debuggers. However, making the $\mathbb{K}$ debugger function as an implementation specific debugger presents challenges.

5.2 CHALLENGES AND SOLUTIONS

In $\mathbb{K}$ debugger, given the generality of its operation, encounters certain problems that need to be addressed in order to provide a familiar debugging experience. In the remainder of this chpater, we mention the problems, and their solutions that allow the debugger. to manage complexity arising out of generality.

5.2.1 Performance

A debugging session, depending on the target language's semantics and program being executed, can have many thousand number of steps (see chapter 7). As mentioned in chapter 6, the architecture of the $\mathbb{K}$ framework is such that the backend can perform one-shot execution until a specified number of steps from the current term has been taken. Since the debugger allows stepping forward, backwards, and to any specific term in the trace, it needs a mechanism to store states that have been already encountered. Algorithm 5.1 describes the naive approach to the issue of storing states.

Algorithm 5.1 Simple Algorithm

Require: φ_0, n, where φ_0 is the initial pattern, and n is the number of steps to take. i is the current step, i.e. φ_i represents pattern is trace observed at step i. φ_0 is stored in the current set of states.

$i \leftarrow 0$
while i $\leq n$ **do**
  $\varphi_{i+1} \leftarrow$ runBackend$(\varphi_i, 1)$
  debuggerState $\leftarrow \varphi_i$
  $i \leftarrow i + 1$
end while

The simple algorithm works well for small programs, but for larger programs, the approach fails due to the following -

- The call to *runBackend* presents a performance bottleneck. Large programs using the

simple algorithm will make n calls to the backend, which presents significant performance issues.

- As the number of patterns stored increases, the memory usage the debugger rises. For large enough programs, execessive memory usage makes will make the debugger unusable.

In order to get around the performance related issues, the $\mathbb{K}$ debugger uses a checkpoint based approach that records patterns at given intervals.

Algorithm 5.2 Generalized Checkpoint Based Step Algorithm

Require: φ_0, n, where φ_0 is the initial pattern, and n is the number of steps to take. i is the current step, i.e. φ_i represents pattern is trace observed at step i. φ_0 is stored in the current set of states. c is the checkpoint interval.

$i \leftarrow 0$

while $\lfloor \mathrm{i} \times c \rfloor \leq n$ **do**

$\varphi_{(i \times c)} \leftarrow \mathrm{runBackend}(\varphi_i, c)$

$\mathrm{debuggerState} \leftarrow \varphi_{(i \times c)}$

$i \leftarrow i + 1$

end while

Algorithm 5.2 reduces the calls to *runBackend* by a factor of c, which is the checkpoint interval. The default checkpoint interval employed in the debugger is 50. However, the user can adopt an interval to change the checkpoint interval in accordance with the program being debugged. Note that the simple algorithm is a special case of the Generalized Checkpoint Based Step Algorithm, with a checkpoint size of 1.

The Generalized Checkpoint Based Step Algorithm improves performance. However, the use of the algorithm requires tweaks to the trace exploration. We present the trace exploration algorithm used by the debugger.

Algorithm 5.3 Trace Exploration Algorithm

Require: Given a debugger session with state containing trace $\varphi_0 \Rightarrow \varphi c \Rightarrow \varphi_{2\times c} \Rightarrow \cdots \Rightarrow \varphi_{n'\times c} \Rightarrow \cdots \Rightarrow \varphi_f$ where $n' \times c \leq n$ (total number of steps taken in the trace so far). Let j be the step in the trace the user wants to access, assuming $0 < j$. checkpointStep is the checkpoint based step algorithm described in algorithm 5.2

if $\varphi_j \in$ debuggerState.checkpoints **then**
  debuggerState.currentPattern $\leftarrow \varphi_j$
  if $j < f$ **then**
    $\varphi_{temp} \leftarrow$ debuggerState.checkpoints$[\lfloor \frac{j}{c} \rfloor]$
    debuggerState.currentPattern $\leftarrow$ runBackend$(\varphi_{temp}, (f - \lfloor \frac{j}{c} \rfloor))$
  end if
else
  $\varphi_{temp} \leftarrow \varphi_f$
  debuggerState $\leftarrow$ checkpointStep$(\varphi_{temp}, (j - f))$
end if

5.2.2 Large Patterns

$\mathbb{K}$ configurations become large as the complexity of language being defined increases. For instance, the configuration of the C semantics has over a 100 cells. Moreover, when large programs are executed, the contents of the cells can make patterns observed in the trace very large.

In order to make configuration management easier, the debugger uses the following strategies -

- At the start of every debug session, the patterns observed are not printed unless the user explicitly specifies so. The debugger's *peek* command prints the entire configuration of the debugger's state's current pattern.

- In order to allow finer grained printing than the *peek* command, the *show* command can be used. The show command takes as input a pattern φ_i, and matches the pattern against the current configuration. It then prints the result of the match. For instance, if the user would like to only view the contents of the $\langle k \rangle$, the command *show* $\langle k \rangle X \langle / k \rangle$ would be useful. The debugger would ask the backned to solve the ML implication pattern, $\Psi \equiv (\langle k \rangle X \langle / k \rangle \implies \varphi_i)$, and prints the result. In this partiular case, the result would be something of the form $X->$ pgm, where pgm is the concrete program AST.

- In order to further aid configuration printing, the *watch* commamnd can be used. A watch is a pattern φ_w that's matched every time the debugger's state's current pattern changes, and the result is printed. For instance, if the user adds the pattern $\varphi_{w'} \equiv \langle k \rangle X \langle / k \rangle$ as a *watch*, then the implication $\Psi_w \varphi_{w'} \equiv \langle k \rangle X \langle / k \rangle \implies \varphi_{current}$ will be solved whenever the state changes. Thus, in this example, the contents of the k cell will be printed on every step. Since the k cell's contents are, by convention, the program being run, this watch allows the debugger to effectively funciton as a traditional program debugger (with a step command), with the exception that the steps in the $\mathbb{K}$ debugger's case are semantics rewrites. Note that since traditional debuggers don't use formal langauge semantics as their foundational bases, their step commands may also lack clear semantics. This provides the $\mathbb{K}$ debugger a distinct advantage over traditional language specific debuggers, without compromising on the usability of the tool. These solutions buttress our claim that the $\mathbb{K}$ debugger is a viable formal alternative to traditional debuggers. Note, that the *show*, and *watch* commands allow the use of configuration concretization ability of $\mathbb{K}$, which makes specifying watch patterns very simple.

5.2.3 Concretizing Generality for Target Language

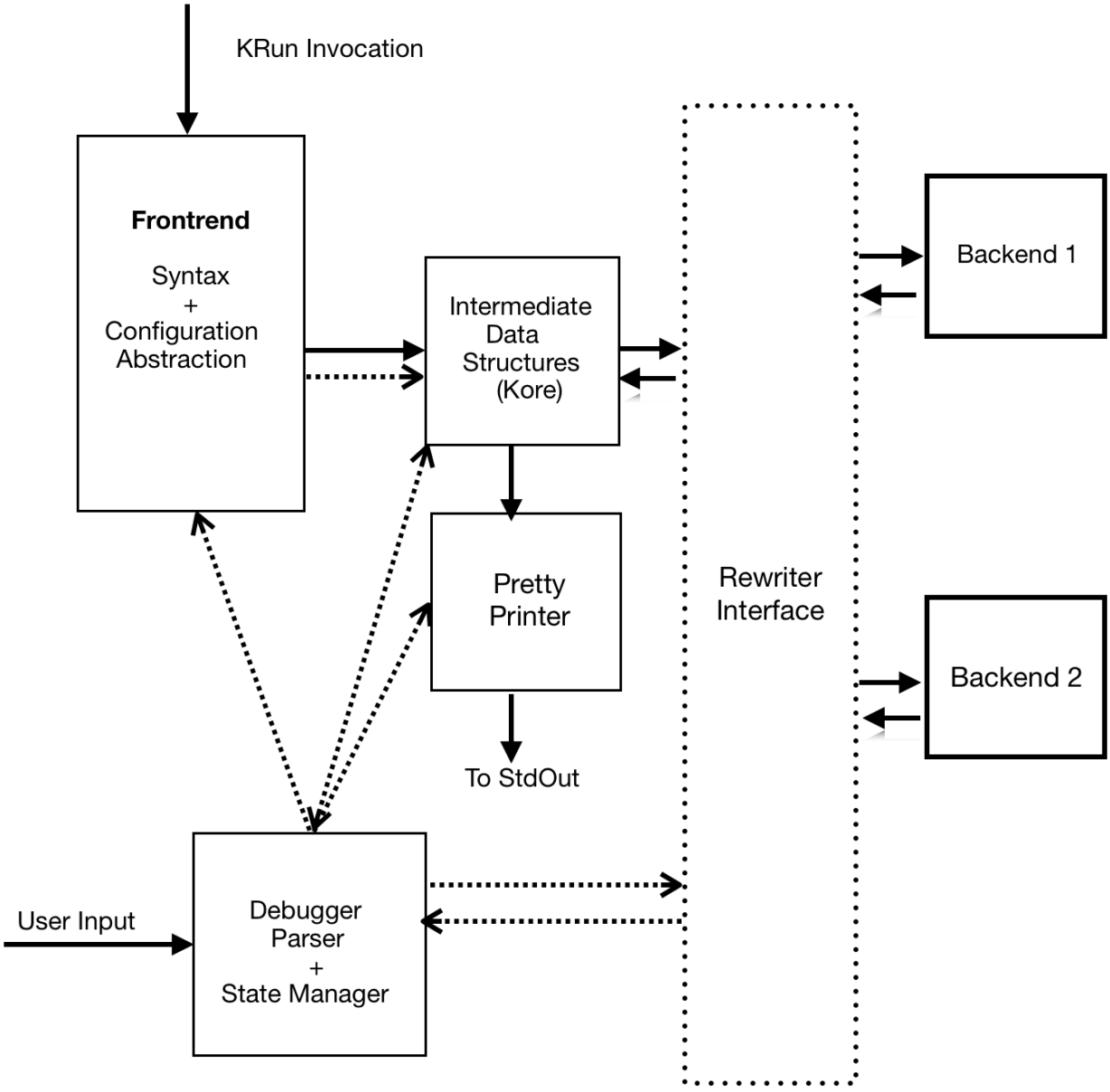
Since the $\mathbb{K}$ debugger is language-independent, it's more general than language-dependent debuggers. However, the debugger's behavior can be made language specific via the use of its commands. It's untenable however, to enter commands to configure the debugger every time a debug session is started. To solve the issue, the debugger allows user to use its scripting language to write the configuration script ina file, and then use the *source* command to configure the script. The syntax of the debugger's scripting language is exactly the same as the command syntax. The script must contain configuration commands the user would like separated via newlines. The formal syntax of the $\mathbb{K}$ debugger's commands is given in section 3.2.

The user needs to make the behavior of the debugger specific to the target language once. The script can then be reused in a debug session for any program in the language.

Chapter 6: ARCHITECTURE AND CODE

This chapter presents the overall architecture of the $\mathbb{K}$ debugger in context of the $\mathbb{K}$ framework.

Figure 6.1: Architecture of the $\mathbb{K}$ debugger



6.1 OVERVIEW

In figure 6, an overview of the overall architecure is provided. The solid lines in the figure represent control flow of regular `krun` session, while the the dotted lines represent control flow of a `debugger` session.

6.1.1 $\mathbb{K}$ Pipeline Preliminaries

This section briefly touches on the architecture of the $\mathbb{K}$. As seen in figure 6, the main $\mathbb{K}$ architecure can be divided three components.

1. **Frontend** - The frontend consists of the parser for $\mathbb{K}$ syntax, and logic for features of $\mathbb{K}$ such as configuration abstraction, evaluations strategies, and well-formedness checks. For instance, when a $\mathbb{K}$ definition is parsed, the frontend is responsible for completing the parts of pattens that can be inferred from context, like filling in the requisite information for the ... parts of the configuration, and generating heating and cooling rules from strictness attributes.

2. **Kernel** - The $\mathbb{K}$ kernel manages interaction between the frontend and the different $\mathbb{K}$ backends. The frontend parses patterns into data structures, which are shared across the framework, called the Kore data strucutures. The kernel uses the kore patterns given to it via the frontend, and calls passes them to one of the rewriter backends, based on user input. Then, it takes the output from the backend, and passes it to its Kore Pretty Printer (which also operates on the Kore Pretty Printer)

3. **Backends** - The framework has been designed to support multiple rewriter Backends. At the time of writing, the two main supported backends by $\mathbb{K}$ were the Java and Ocaml backends. The Java backend is specialized for symbolic execution, while the Ocaml backend supports fast concrete execution.

6.2 $\mathbb{K}$ DEBUGGER PIPELINE

6.2.1 Requirements

The $\mathbb{K}$ debugger's pipeline was designed to obtain certain goals.

- **Multiple Backends Support** - The debugger, unlike some $\mathbb{K}$ tools like the $\mathbb{K}$ prover, was designed to work with all $\mathbb{K}$ backends.

- **Minimal Dependence on Rewriters** - The debugger has to minimize calls to the rewriter, for performance.

6.2.2 Implementation Details

The debugger's pipeline addresses requirements mentioned in 6.2.1 in the following ways
-

- **Rewriter Interface** - The $\mathbb{K}$ debugger only interacts with various $\mathbb{K}$ backends using the rewriter interface. Any backend willing to support the debugger, must in honor the interface. This allows the debugger to stay agnostic to the backend being used to perform rewriter, which makes it independent of the backend.

  Furthermore, this weak coupling between backends the debugger also mean that the backends can be updated and maintained indpendently of the debugger.

- **Independence from Rewriters** - In order to maintain complete decompling from Rewriters, The debugger maintains its own state. This further allows the debugger to query the rewriters only when needed.

Debugger Control Flow

This section goes into depth about the debugger's control flow, as shown in fig 6. At the start of a debug session, the program to be debugged is already loaded. The loading of the program happens via the regular `krun` pipeline. The debugger's parser then accepts commands in its syntax, and performs actions accordingly. Note that since some of the commands, like `watch` use $\mathbb{K}$ syntax, they're first parsed using $\mathbb{K}$'s regular syntax parser, then passed throught $\mathbb{K}$'s compilation pipeline, which involves steps like configuration concretization. The processed patterns are then returned to the debugger. The debugger only functions over the generic "kore" data structures of $\mathbb{K}$. $\mathbb{K}$ includes a Pretty Printer for ML patterns, which is shared by both the normal and debug pipelines. Information presented to the user is passed via the Pretty Printer, which allows the debugger to remain decoupled from umparsing the patterns. Most importantly, the debugger interacts with the rewrite engines via the "rewriter" interface. In order to support debugging, the backend must implement the "rewriter" interface. The interface provides a coherent mechanism of communication between the backend and the debugger. The task of implementing a new backend and making it work with the debugger is also becomes simpler, since the backend

needs to only honor the "rewriter" interface, and would work with the debugger without any change to the debugger's code itself.

Chapter 7: CONCRETE APPLICATIONS

This section desribes the usage of the $\mathbb{K}$ debugger in developing a the model of a non-trivial language. We choose a real-world language to demonstrate the debugger's applications to support our overall claim the our work allows for bypassing the use conventional debuggers in favor of our language-independent and formal approach. The language we chose to demonstrate the usability of the debugger is the EVM [14].

7.1 OVERVIEW

This section briefly presents an overview of the semantics body we intend to use as the driving force behind the applications section.

7.2 DEBUGGING METHODOLOGY

We present an outline of what debugging both the semantics and the prgram using the $\mathbb{K}$ debugger. Consider a correct program execution trace, denoted by $\tau_{correct} \equiv \varphi_0 \Rightarrow \varphi_1 \Rightarrow \cdots \Rightarrow \varphi_f$, where φ_0 is the initial state, and φ_f, is the final correct state. The presence of the bug is indicated by the presence of a pattern, say φ_i, s.t. that behavior of the program diverges from the expected behavior at that point. More formally, let the observed trace, which we known to be incorrect be $\tau_{incorrect} \equiv \varphi_0 \Rightarrow \varphi_{1'} \Rightarrow \cdots \Rightarrow \varphi_{f'}$. Given the incorrect trace, one needs to localize pattern $\varphi_{i'}$ in $\tau_{incorrect}$ s.t. $\tau_{correct} \restriction \varphi_{i-1} \equiv \tau_{incorrect} \restriction \varphi_{i-1}$. Here, $\tau \restriction \varphi$ represents trace τ restricted to pattern φ, and $\tau_{correct}$ is the desired correct trace. Using the debugger trace exploration commands, one needs to first find pattern φ_i. Once φ_i is localized, the debugger commmands for analyzing φ_i can be used to detect the issue. In the case of a bug in the semantics, usually the rewrite rule, $\psi \equiv \varphi_\psi \rightarrow \gamma_\psi$ hast to be fixed. More formally, we need to find the rule $\psi' \equiv \varphi_{\psi'} \rightarrow \gamma_{\psi'}$ s.t. after the rule application, $\Psi_{correct} \equiv (\tau_{correct} \restriction \varphi_i \equiv \tau_{incorrect} \restriction \varphi_i)$ holds. In the case of the semantics being correct, the pattern φ_i itself is incorrect. The pattern must then be rectified so that $\Psi_{correct}$ holds. The debugging is performed until there exist no pattern φ_i, such that $\tau_{correct} \restriction \varphi_i \not\equiv \tau_{incorrect} \restriction \varphi_i$. Simply put, the session is considered complete when the deviant trace is the same as the expected trace.

7.3 EVM SEMANTICS

The EVM is a quasi Turing complete stack based language that forms the basis of programmable transactions on the Ethereum Network. The $\mathbb{K}$ semantics of the EVM were developed after the network suffred a series of security related issues, highlighting the need for formal analysis tools. Thus, the $\mathbb{K}$ semantics of the EVM were devloped using the official informal specification of the language (also referred to the yellowpaper).

Since the EVM is assembly-like, an EVM program consists of opcodes (operations) followed by data. For instance, "PUSH" in EVM is an opcode that pushes the following 32-bytes in the program binary on to the stack. What separates the EVM from traditional assembly languages is that it's not meant to be executed on specialized hardware. Instead it's meant to be simple enough so that it can be interpreted on nodes running on the Ethereum network. The EVM also has an underlying concept of "gas", or the amount of computation resources available to the program for execution. Gas is provided to the program by the Ethereum transaction that initiates the call to the program. The concept of gas makes the EVM highly non-trivial, as nodes need to support operations like reverting to an older state if the execution of the program runs out of gas. Moreover, the EVM also has different exceptional states, which nodes are expected to handle.

An evm program is a map between program counter values, and opcodes. The $\mathbb{K}$ semantics of the EVM mimic the official yellowpaper, but since the $\mathbb{K}$ semantics are executable, they can be actually used to execute programs. Moreover, the $\mathbb{K}$ debugger can be used to view how the program is transformed using the formal semantics.

Completeness of the semantics

Since the $\mathbb{K}$ specification of the semantics of the EVM results in an interpreter, it's possible to run EVM programs. The claim behind the completeness of the semantics is based on the fact that the automatically generated interpreter passes all the conformance tests in the offical Ethereum Clent conformance test suite. The development of the seantics were driven via the $\mathbb{K}$ debugger. Given the conformance tests, the authors of the semantics were able to use the debugger to fix the semantics using the tests. Once the semantics were done, the debugger forms a part of the tools that function on the semantics to run, debug and formally analyze EVM programs.

7.3.1 Developing semantics using the $\mathbb{K}$ debugger

The $\mathbb{K}$ debugger proved to be an enormously useful tool in the development of the semantics. The test driven approach followed in the development of the semantics resulted in the availability of a rich body of programs with known behavior. Whenever a test during the devlopment cycle of the semantics failed, the result of the failure was assumed to be a subtle fault in the semantics. The other possibility was a bug test, which was rare as the tests were conformance tests, and were expected to be correct. Nevertheless, during the devleopment of the semantics, bugs in both the semantics and the tests themselves were discovered. The $\mathbb{K}$ debugger proved to be useful in both failure cases.

7.3.2 Concrete Debug Session

In order to demonstrate the usage of the debugger, this section presents a sample run of an EVM program in the debug mode, from section 5.1 of [14]. In the concrete example, some details of the the EVM configuration, that aren't required for the evaluation of the tool, are omitted for readability. Note that we use a real-world example, instead of a toy language to emphasize the fact the debugger can be used in practical, real world scenarios.

The debugger session is started with using a evm program from the conformance test, and then use the `jump 91` command to jump to an interesting pattern in the trace.

Example 7.1.

```
KDebug> jump 91
Jumped to Step Number 91
KDebug> peek
...
    <op> #next ~> #execute </op> ...
    <program> ... 33 |-> PUSH ( 32 , N ) ... </program> ...
    <wordStack> N : ... </wordStack> ...
    <pc> 33 </pc>
    <gas> 99997 </gas>
...
```

In example 7.1, the $\langle op \rangle$ cell contains the next instrction to execute to execute. Instruction beginning with the letter "#" are internal to the semantics. For instance, at the top of the $\langle op \rangle$ cell, the $\#next$ instruction is present, which in the semantics indicates readiness for execution of the instruction at the program counter value in the $\langle pc \rangle$ cell.

Example 7.2.

```
KDebug> step
1 Step(s) Taken.
KDebug> peek
...
    <op>
        #pushCallStack
          ~> #exceptional? [ PUSH ( 32 , N ) ]
          ~> #exec         [ PUSH ( 32 , N ) ]
          ~> #pc           [ PUSH ( 32 , N ) ]
          ~> #? #dropCallStack : #popCallStack ~> #exception ?#
          ~> #execute
    </op> ...
    <program> ... </program> ...
    <wordStack> N : ... </wordStack> ...
    <pc> 33 </pc>
    <gas> 99997 </gas>
...
```

Example 7.2 shows taking one more step: the next instruction is loaded, and ready to be executed. Note the `#exceptional?` instruction - it's an internal instruction that checks whether the execution of the opcode itself can result in an exception. This check exists due to the fact that the EVM only allows execution of an opcode if it doesn't lead to an exception. The "peek" command here demonstrates precisely how the program changes according to the semantics. If the exceptional check succeeds, the `#exec` opcode executes the program, and the `#pc` opcode changes the program counter value, completing the execution of one EVM cycle.

It takes 16 steps from the pattern above to reach another pattern where the execution for once cycle is completed. The level of detail provided by the $\mathbb{K}$ debugger, due to its SOS foundations is higher than traditional ad-hoc debugger. The extra detail may seem unneccessary for debugging simple bugs, and may lead one to believe that the $\mathbb{K}$ debugger should only be used in cases where traditional debuggers fail. However, as mentioned in section 5.2.3, the granularity can be adjusted to address the problem of excessive detail. For instance, in this particular case, one can choose to always take 18 steps instead of 1, to synchronize the step cycle of the debugger with that of the EVM semantics, and bypass

extra details present between the steps. Moreover, the user can set watches to only show the parts of the configuration that are necessary, making the debug session very specific to not just the target language (EVM in this instance) but the program itself. The intention, of using this example was to emphasize that the $\mathbb{K}$ debugger is intended to not "augment", but complete replace traditional debuggers ad-hoc debuggers.

Chapter 8: CONCLUSION AND FUTURE WORK

8.1 CONCLUSION

This work presents an alternative to traditional language dependent based approaches to debugging. Our approach of langugage independent debugging has distinct dvantages over traditional approaches.

8.1.1 Language Independence

Our approach is more general than traditional approaches, and works for any programming language, as long as the semantics of the language is captured using $\mathbb{K}$. It requires no extra implmentation or maintainence effort. Furthermore, it provides a coherent platform for debugging. Users need not learn different debugging approaches for different languages. Our approach is general, and can be specialized to fit the need of the target language. We believe this allows our debugger to be a viable alternative to traditional debuggers.

8.1.2 Formal Semantics Based

Our approach is formal, and based on Reachability Logic. The $\mathbb{K}$ semantics of the language ared used as axioms for RL proofs in the framework. Hence, we can capture subtle bugs that traditional debuggers, which are based on ad-hoc semantics of the language miss. Note that the added layer of detail due to the SOS foundations can be managed via the debugger's commands.

This work also presents details of the debugger's implementation. We present the architecture of our debugger, with details about how it's decoupled from other components of the framework.

Finally, we also demonstrate, with a real world example how the debugger's performance, and usability can be employed to develop a non trivial language. Our debugger performs favourably, and comes across as a useful tool. Since the debugger has already been tried on practical large languages, we believe that it can replace traditional language specific debuggers.

8.2 FUTURE WORK

We identify the following areas where the debugger needs further work -

- **Multiple Debug Sessions** - Currently, the debugger can handle only one a single session at a time. Adding the ability to debug multiple sessions at the same time is desirable.

- **Formal Proof Objects** - The $\mathbb{K}$ debugger has its theoretical foundations in RL and ML. However, it at the moment does not is based in the produce ML artificacts that can idependently check the debugger's actions. For instance, the watch statement involves performing an ML match of the watch pattern on the current configuration. An ML match is formal operation, and we can use the proof system of ML to justify the match. Producing proof objects or artificacts that can be checked independently of the debugger would provide more confidence about the correctness of the debugger's actions.

REFERENCES

[1] GNU Free Software Foundation, "Gdb: The gnu project debugger," //https://www.gnu.org/software/gdb/.org.

[2] Orcle, Inc., "JDB: The Java Debugger," https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html.

[3] International Organization for Standardization, "ISO/IEC 9899:2011," https://www.iso.org/standard/57853.html.

[4] Valgrind Developers, "Valgrind," https://www.valgrind.org.

[5] Runtime Verification, Inc., "RV-Match," https://runtimeverification.com/match/.

[6] D. Guth, C. Hathhorn, M. Saxena, and G. Rosu, "Rv-match: Practical semantics-based program analysis," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, ser. LNCS, vol. 9779. Springer, July 2016, pp. 447–453.

[7] Formal Systems Lab, UIUC, "The K framework," 2006, http://kframework.org.

[8] C. Ellison and G. Roşu, "A formal semantics of C with applications," University of Illinois, Tech. Rep. http://hdl.handle.net/2142/17414, November 2010.

[9] D. Bogdănaş and G. Roşu, "K-Java: A Complete Semantics of Java," in *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*. ACM, January 2015, pp. 445–456.

[10] D. Park, A. Ştefănescu, and G. Roşu, "KJS: A complete formal semantics of JavaScript," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, June 2015, pp. 346–356.

[11] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, B. Moore, Y. Zhang, D. Park, and G. Roşu, "Kevm: A complete semantics of the ethereum virtual machine," in *Computer Security Foundations Symposium*, 2018.

[12] G. Rosu and T. F. Serbanuta, "K overview and simple case study," in *Proceedings of International K Workshop (K'11)*, ser. ENTCS, vol. 304. Elsevier, June 2014, pp. 3–56.

[13] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[14] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Roşu, "Kevm: A complete semantics of the ethereum virtual machine," in *Technical Report*, 2018.

[15] A. Stefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu, "Semantics-based program verifiers for all languages," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2983990.2984027 pp. 74–91.

[16] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969. [Online]. Available: http://doi.acm.org/10.1145/363235.363259

[17] G. Roşu and A. Ştefănescu, "From hoare logic to matching logic reachability," in *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, ser. LNCS, vol. 7436. Springer, Aug 2012, pp. 387–402.

[18] G. Roşu, "Matching logic," *Logical Methods in Computer Science*, vol. 13, no. 4, pp. 1–61, December 2017.

[19] Oracle, Inc., "JavaCC - The Parser Generator," https://www.javacc.org.

[20] Runtime Verification, Inc., "IELE: Semantics of New Cryptocurrency VM in K," https://www.github.com/runtimeverification/iele.